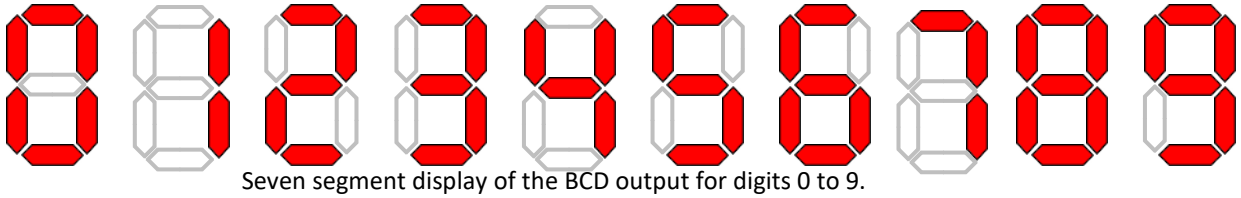
UNIVERSITY OF MANITOBA
Department of Electrical and Computer Engineering

2020 Fall - ECE 2220 Laboratory 3

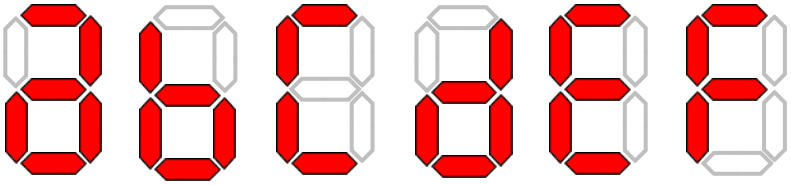
Binary Code Decimal, Hexadecimal Number Representation

1) Introduction

In this lab the seven segment display (SSD) will be used to output the results of a 4-bit binary word in hexadecimal and then binary coded decimal output. A binary-coded-decimal (BCD) or a hexadecimal to SSD converter display is simply a combinational circuit with 4 binary inputs and 7 outputs. The 4-bit input is the BCD representation of digits 0-9 and the hexadecimal representation of digits 0-F and the. The 7-bit output is the state of each of SSD segments as shown in see in the figure below. Note the representation of some digits like 1, 6, and 9 might be different from one decoder to another.

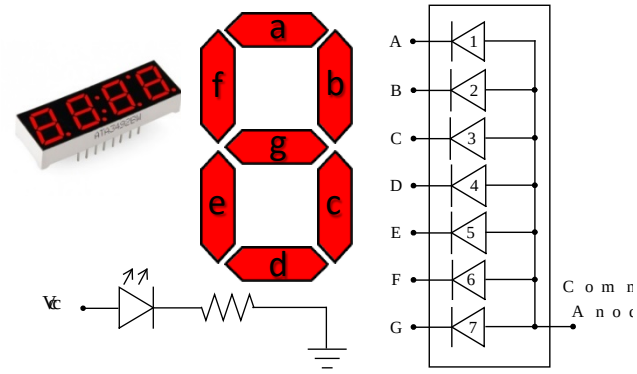


Seven segment display of the BCD output for digits 0 to 9.



Seven segment display of the HEX output for digits 0 to F.

Seven-segment displays (SSD) are commonly found on computers, watches, VCRs, and other electronic devices to display numbers and characters. The seven-segment displays in the lab consist of seven Light Emitting Diodes (LEDs) in the configuration of a number “8”. Different segments can be illuminated to display different numbers and letters. The segments of a seven-segment display are illustrated in the figure below. The SSDs, in general, come in packages with either a common anode or a common cathode. The SSDs on the DE10 board are common-anode. In this format the LED turns on with negative logic – i.e. low.



Common-anode SSD display

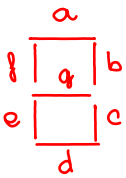
2020 Fall - ECE 2220 Laboratory 3

Binary Code Decimal, Hexadecimal Number Representation

2) Instructions

Take a 4 bit input from the switches SW[3], SW[2], SW[1], and SW[0]. Display the binary number as a BCD number on right most seven segment display HEX[0]. If the binary input number is greater than 9, the letter E should be displayed.

a) Create a truth table and Karnaugh maps for the BCD-seven segment display



Decimal	BCD – input switches				Segment (LSB → MSB)							hex
	s3	s2	s1	s0	a	b	c	d	e	f	g	
0	0	0	0	0	0	0	0	0	0	0	1	40
1	0	0	0	1	1	0	0	1	1	1	1	79
2	0	0	1	0	0	0	1	0	0	1	0	24
3	0	0	1	1	0	0	0	0	1	1	0	30
4	0	1	0	0	1	0	0	1	1	0	0	19
5	0	1	0	1	0	1	0	0	1	0	0	12
6	0	1	1	0	0	1	0	0	0	0	0	02
7	0	1	1	1	0	0	0	1	1	1	1	78
8	1	0	0	0	0	0	0	0	0	0	0	00
9	1	0	0	1	0	0	0	0	1	0	0	18
10	1	0	1	0	0	0	0	0	0	1	0	20
11	1	0	1	1	1	1	0	0	0	0	0	03
12	1	1	0	0	0	1	1	0	0	0	1	46
13	1	1	0	1	1	0	0	0	0	1	0	21
14	1	1	1	0	0	1	1	0	0	0	0	06
15	1	1	1	1	0	1	1	1	0	0	0	0E

1 var → 8 of 1s
2 vars → 4 of 1s
3 vars → 2 of 1s
4 vars → 1 of 1

AND ·
OR +

a

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	0	1	0	0
01	1	0	0	0
11	0	0	0	0
10	0	0	0	0

$a =$ $S_3' S_2' S_1' S_0 + S_3' S_2 S_1' S_0'$

b

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	0	0	0	0
01	0	1	0	1
11	1	1	1	1
10	0	0	1	1

$b =$ $S_3' S_2 S_1' S_0 + S_3' S_2 S_1 S_0' + S_3 S_2 + S_3 S_1$
$= S_3' S_2 (S_1 \oplus S_0) + S_3 S_2 + S_3 S_1$

c

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	0	0	0	1
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$c =$ $S_3' S_2' S_1 S_0' + S_3 S_2 + S_3 S_1$

d

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	0	1	0	0
01	1	0	1	0
11	0	0	0	0
10	0	0	0	0

$d =$ $S_3' S_2' S_1' S_0 + S_3' S_2 S_1' S_0' + S_3' S_2 S_1 S_0$
$= S_3' S_1' (S_2 \oplus S_0) + S_3' S_2 S_1 S_0$

e

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	0	1	1	0
01	1	1	1	0
11	0	0	0	0
10	0	1	0	0

$e =$ $S_3' S_0 + S_3' S_2 S_1' S_0 + S_3 S_2' S_1 S_0$

f

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	0	1	1	1
01	0	0	1	0
11	0	0	0	0
10	0	0	0	0

$f =$ $S_3' S_2' S_0 + S_3' S_2' S_1 + S_3' S_1 S_0$
$= S_3' S_2' (S_0 + S_1) + S_3' S_1 (S_2' + S_0)$

g

$S_3 S_2$ \ $S_1 S_0$	00	01	11	10
00	1	1	0	0
01	0	0	1	0
11	0	0	0	0
10	0	0	0	0

$g =$ $S_3' S_2' S_1' + S_3' S_2 S_1 S_0$

b) Create the Verilog code to perform this BCD function and output using simple combinational logic primitives (i.e. AND (&) , OR (|), NOT (~) etc.) Compile your code and download it to the DE-10 board. K-map a, b, c, d, e, f, g 7 cases

c) Re-write this code using a "case" statement. Compile your code and download it to the DE-10 board. 0-9, E 16 cases

d) Again using a case statement, re-write the code to display the hexadecimal representations of the input switches. Comment on the two Verilog implementations. 0-9, a-f

The Case Statement

The case statement compares an expression to a series of cases and executes the statement or statement group associated with the first matching case:

- case statement supports single or multiple statements.
- Group multiple statements using begin and end keywords.

Syntax of a case statement look as shown below.

```
case ()
< case1 > : < statement >
< case2 > : < statement >
.....
default : < statement >
endcase
```

Example- case

```
1  module mux (a,b,c,d,sel,y);
2  input a, b, c, d;
3  input [1:0] sel;
4  output y;
5
6  reg y;
7
8  always @ (a or b or c or d or sel)
9  case (sel)
10   0 : y = a;
11   1 : y = b;
12   2 : y = c;
13   3 : y = d;
14   default : $display("Error in SEL");
15 endcase
16
17 endmodule
```

Example- case without default

```
1  module mux_without_default (a,b,c,d,sel,y);
2  input a, b, c, d;
3  input [1:0] sel;
4  output y;
5
6  reg y;
7
8  always @ (a or b or c or d or sel)
9  case (sel)
10   0 : y = a;
11   1 : y = b;
12   2 : y = c;
13   3 : y = d;
14   2'bxx,2'bx0,2'bx1,2'b0x,2'b1x,
15   2'bzz,2'bz0,2'bz1,2'b0z,2'b1z : $display("Error in SEL");
16 endcase
17
18 endmodule
```

The example above shows how to specify multiple case items as a single case item.

The Verilog case statement does an identity comparison (like the === operator); one can use the case statement to check for logic x and z values as shown in the example below.

Example- casez

```
module casez_example();
reg [3:0] opcode;
reg [1:0] a,b,c;
reg [1:0] out;

always @ (opcode or a or b or c)
casez(opcode)
  4'b1zzx : begin // Don't care about lower 2:1 bit, bit 0 match with x
              out = a;
              $display("@%0dns 4'b1zzx is selected, opcode %b",$time,opcode);
            end
  4'b01?? : begin
              out = b; // bit 1:0 is don't care
              $display("@%0dns 4'b01?? is selected, opcode %b",$time,opcode);
            end
  4'b001? : begin  // bit 0 is don't care
              out = c;
              $display("@%0dns 4'b001? is selected, opcode %b",$time,opcode);
            end
  default : begin
              $display("@%0dns default is selected, opcode %b",$time,opcode);
            end
endcase

// Testbench code goes here
always #2  a = $random;
always #2  b = $random;
always #2  c = $random;

initial begin
  opcode = 0;
  #2  opcode = 4'b101x;
  #2  opcode = 4'b0101;
  #2  opcode = 4'b0010;
  #2  opcode = 4'b0000;
  #2  $finish;
end

endmodule
```

Example- case with x and z

```
module case_xz(enable);
input enable;

always @ (enable)
case(enable)
   1'bz : $display ("enable is floating");
   1'bx : $display ("enable is unknown");
   default : $display ("enable is %b",enable);
endcase

endmodule
```

You could download file case_xz.v here

The casez and casex statement

Special versions of the case statement allow the x ad z logic values to be used as "don't care":

- casez : Treats z as don't care.
- casex : Treats x and z as don't care.

Example- casex

```
1  module casex_example();
2  reg [3:0] opcode;
3  reg [1:0] a,b,c;
4  reg [1:0] out;
5
6  always @ (opcode or a or b or c)
7  casex(opcode)
8    4'b1zzx : begin // Don't care  2:0 bits
9                out = a;
10               $display("@%0dns 4'b1zzx is selected, opcode %b",$time,opcode);
11             end
12   4'b01?? : begin // bit 1:0 is don't care
13               out = b;
14               $display("@%0dns 4'b01?? is selected, opcode %b",$time,opcode);
15             end
16   4'b001? : begin // bit 0 is don't care
17               out = c;
18               $display("@%0dns 4'b001? is selected, opcode %b",$time,opcode);
19             end
20   default : begin
21               $display("@%0dns default is selected, opcode %b",$time,opcode);
22             end
23 endcase
24
25 // Testbench code goes here
26 always  #2  a = $random;
27 always  #2  b = $random;
28 always  #2  c = $random;
29
30 initial  begin
31   opcode = 0;
32   #2  opcode = 4'b101x;
33   #2  opcode = 4'b0101;
34   #2  opcode = 4'b0010;
35   #2  opcode = 4'b0000;
36   #2  $finish;
37 end
38
39 endmodule
```

Example- Comparing case, casex, casez

```
 1 module case_compare;
 2
 3 reg sel;
 4
 5 initial begin
 6   #1  $display ("\n    Driving 0");
 7   sel = 0;
 8   #1  $display ("\n    Driving 1");
 9   sel = 1;
10   #1  $display ("\n    Driving x");
11   sel = 1'bx;
12   #1  $display ("\n    Driving z");
13   sel = 1'bz;
14   #1  $finish;
15 end
16
17 always @ (sel)
18 case (sel)
19   1'b0 : $display("Normal : Logic 0 on sel");
20   1'b1 : $display("Normal : Logic 1 on sel");
21   1'bx : $display("Normal : Logic x on sel");
22   1'bz : $display("Normal : Logic z on sel");
23 endcase
24
25 always @ (sel)
26 casex (sel)
27   1'b0 : $display("CASEX  : Logic 0 on sel");
28   1'b1 : $display("CASEX  : Logic 1 on sel");
29   1'bx : $display("CASEX  : Logic x on sel");
30   1'bz : $display("CASEX  : Logic z on sel");
31 endcase
32
33 always @ (sel)
34 casez (sel)
35   1'b0 : $display("CASEZ  : Logic 0 on sel");
36   1'b1 : $display("CASEZ  : Logic 1 on sel");
37   1'bx : $display("CASEZ  : Logic x on sel");
38   1'bz : $display("CASEZ  : Logic z on sel");
39 endcase
40
41 endmodule
```